SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:19–39 (DOI: 10.1002/spe.493)

Decorating tokens to facilitate recognition of ambiguous language constructs

Brian A. Malloy[1,*,†], Tanton H. Gibbs[1] and James F. Power[2]

[1]*Computer Science Department, Clemson University, Clemson, SC 29634, U.S.A.*
[2]*Department of Computer Science, National University of Ireland, Maynooth, Ireland*

SUMMARY

Software tools are fundamental to the comprehension, analysis, testing and debugging of application systems. A necessary first step in the development of many tools is the construction of a parser front-end that can recognize the implementation language of the system under development. In this paper, we describe our use of token decoration to facilitate recognition of ambiguous language constructs. We apply our approach to the C++ language since its grammar is replete with ambiguous derivations such as the *declaration/expression* and *template-declaration/expression* ambiguity. We describe our implementation of a parser front-end for C++, *keystone*, and we describe our results in decorating tokens for our test suite including the examples from Clause Three of the C++ standard. We are currently exploiting the *keystone* front-end to develop a taxonomy for implementation-based class testing and to reverse-engineer Unified Modeling Language (UML) class diagrams. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: grammar; symbol table; parser; Unified Modeling Language; design pattern; the Facade and Decorator design patterns

1. INTRODUCTION

Software tools are fundamental to the comprehension, analysis, testing and debugging of application systems. Tools can automate repetitive tasks and, with large-scale systems, can enable computation that would be prohibitively time consuming if performed manually. The Java language is well supported with libraries and tools to support application development [1–3]. The lack of tool support for

*Correspondence to: Brian A. Malloy, Computer Science Department, Clemson University, Clemson, SC 29634, U.S.A.
†E-mail: malloy@cs.clemson.edu

Contract/grant sponsor: Enterprise Ireland; contract/grant number: IC-2001-061

Received 9 April 2002
Revised 29 August 2002
Accepted 29 August 2002

Published online 12 November 2002

applications using the C++ language is especially noteworthy; in fact, there is presently no tool available in the public domain that can fully accept applications written in ISO C++ [4].

One explanation for the lack of software tools for C++ is the difficulty in constructing a front-end for the language, as described in [5–10]. This difficulty results, in part, from the complexity and scale of the language. However, a more important problem is the ambiguity inherent in many C++ language constructs [6,7,10,11]. Many C++ constructs cannot be recognized through syntactic considerations alone. For example, the difficulty in distinguishing a declaration from an expression can only be resolved by performing name lookup [6,12]. Given the obvious mapping of objects to the roles in compiler front-end construction such as a parser or scanner, it is unfortunate that the carry-over of object technology to parser development has been minimal. Moreover, we believe that the tools of software engineering can greatly facilitate management of the scale and complexity of parser front-end construction.

In this paper, we describe our use of token decoration to facilitate recognition of ambiguous language constructs. We use the Unified Modeling Language (UML) [13,14], to describe our object-oriented design that exploits design patterns including the Decorator and Facade pattern [15]. A key framework in our system, the TokenDecorator, intercepts tokens and, in some cases, mutates the token; we refer to our technique as *token decoration* after our use of the Decorator pattern.

We apply our approach to the C++ language since its grammar is replete with ambiguous derivations such as the *declaration/expression* and *template-declaration/expression* ambiguity. We describe our implementation of a parser front-end for C++, *keystone*, and our results in decorating tokens for several suites of programs including a suite garnered from Clause Three of the C++ standard [12]; we use the Clause Three suite to compare name lookup in *keystone* to several other important compilers.

There are important advantages in our approach to token decoration. First, token decoration allows us to disambiguate language constructs without modifying the grammar; thus, we can use the grammar listed in Appendix A and described throughout the C++ standard with the modification of only one token. Readers of the C++ standard can correlate references to the grammar in the standard with the grammar in our parser and developers who are familiar with the standard will recognize the grammar in our parser front-end. In contrast, the GNU C++ grammar from *gcc* version 2.96 bears little resemblance to the C++ standard and the corresponding *bison* parser contains over 500 goto statements. A second advantage of our approach is that token decoration is encapsulated in a class framework, which facilitates maintenance of the parser. This modularity permits developers to exploit a scanner or parser generator of their choice; in fact, we have used three different parser generators in our development of *keystone* [8]. A third advantage of our approach is that the encapsulation afforded by the token decorator facilitates the development of an application programmer interface (API) based on the tokens that are intercepted or decorated. We use this low-level API to gather statistics reported in Section 6. A fourth advantage is that our exploitation of object-technology includes a facade into the parser that also permits us to easily construct an API for the parser front-end. We have used this high-level API to reverse engineer class diagrams for C++ applications [16] and to develop a taxonomy for implementation-based class testing [17]. Finally, our token buffer, incorporated into the token decorator, obviates the need for an abstract syntax tree or attaching attributes to the grammar.

In the next section we introduce terminology important to the problem that we address, and describe some of the difficulty in parsing C++, including the template-declaration/expression ambiguity and

argument-dependent name lookup[‡]. In Section 3 we describe token decoration and in Section 4 we present details of our implementation including our solutions to the declaration/expression ambiguity and argument-dependent name lookup. In Section 6 we present our implementation results for a suite of benchmark programs and in Section 7 we describe the work related to our problem. Section 8 concludes the paper.

2. BACKGROUND

In Section 2.1, we define terms and introduce concepts related to the problem that we consider. We describe some of the difficulties involved in the construction of a parser front-end for C++, including the importance of a solution to the name lookup problem. In Section 2.2 we discuss namespaces and the use of qualified names, and in Section 2.3 we describe argument-dependent name lookup. In Section 2.4, we overview our design of a system to construct a symbol table and to perform name lookup for C++. Our research on the design of *keystone* can be found in references [8,18,11]; in this paper we focus on details for decorating and buffering tokens in the *keystone* implementation.

2.1. Terminology and statement of the problem

Software developers need tools to facilitate the design, analysis and testing of the system under development. Many useful tools require a parser front-end to compute and store information about the program, and to perform the analysis needed. A *parser front-end* performs lexical and syntactic analysis, constructs a symbol table, performs semantic analysis and possibly generates an intermediate representation of the program. A *symbol table* is a data structure that stores information about the types and names used in the program.

Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars (CFGs) [19]. Most of the constructs of languages such as Pascal and Ada can be specified by CFGs, and parser front-ends for these languages can base their recognition on syntactic considerations alone. An exception to this easy-parse rule can be found in the language C, where a declaration may not be easily distinguished from an expression. Consider the following code segment:

```
f (x);
```
(1)

Intuitively, the above segment appears to be an expression involving an invocation of function `f` with parameter `x`. However, if the context includes the declaration

```
typedef int f;
```
(2)

then the code segment is actually a declaration of `x` as an integer variable with redundant parentheses. This *declaration/expression ambiguity* not withstanding, parser front-ends for the C language have not been difficult to construct. However, a parser front-end for the C++ language has proven elusive and the difficulties involved have been described in [5–7,9–11]. Currently, there is no parser front-end

[‡]Argument-dependent name lookup is also referred to as Koenig lookup, named after Andrew Koenig, who established the definition and is a longtime member of the C++ standards committee.

```
( 1)  namespace A {
( 2)    namespace B {
( 3)      namespace C { int x; }
( 4)    }
( 4)  }
( 5)  A::B::C::x = 0;
```

Figure 1. Three namespaces, with a declaration of x local to the inner namespace C. Line 5 contains a line of code to access x through qualification.

in the public domain that can fully parse the language described in the ISO C++ standard [4]. Many constructs in the C++ language cannot be recognized by syntactic consideration alone; these constructs not only include the *typedef* declaration/expression ambiguity of C, but the ISO C++ grammar also includes context-dependent keywords for *namespace*, *class*, *enumeration* and *template* declarations [12, Appendix A].

To illustrate the declaration/expression ambiguity introduced by templates into C++, consider that for a code segment beginning '`a < b` ...', the name `a` must be looked up to determine whether the $<$ is the beginning of a template argument list or a less-than operator [12, Section 3.4.5/1]. Thus, the disambiguation of many C++ constructs requires a solution to the name lookup problem. The *name lookup* problem is defined as follows: given the use of a name in a program, find the corresponding declaration of that name.

The GNU Free Software Foundation offers *gcc*, a public domain compiler for C++ and other languages. However, it is difficult to de-couple the parser front-end of *gcc* from the compiler internals. Even if de-coupling were achieved, low-level access to the internals of *gcc* is not easily accomplished. Moreover, *gcc* does not, at present, fully parse the language described in the ISO C++ standard.

2.2. Namespaces and qualified names in C++

Namespaces act as a modularization construct in C++, allowing the programmer to partition the names used in a program to prevent them from interfering with each other. Thus, given variable `x` declared in namespace `A`, once outside namespace `A` we may refer to the variable using explicit qualification, as in `A::x`.

Namespaces may be nested, in which case name occurrences inside the inner namespace may refer to those already declared at the outer level without the need for qualification, with this process continuing recursively, eventually reaching the global namespace where all namespaces are ultimately nested. Figure 1 illustrates local variable `x` in namespace `C`, nested in namespace `B`, nested in namespace `A`. We can access the variable `x` declared in namespace `C` with the qualification `A::B::C::x`, as illustrated on line 5 of Figure 1. We revisit this example in Section 4.2, including implementation details about qualified name lookup in *keystone*.

In addition to this textual relationship achieved through qualification or nesting, we may establish a logical relationship between namespaces by importing one into another with a *using* directive. The declarations in a namespace that are imported in this way are treated as though they were originally


```
( 1)  namespace NS {
( 2)    class T { };
( 3)    void f(T);
( 4)  }
( 5)  NS::T parm;
( 6)  int main() {
( 7)    f(parm);
( 8)  }
```

Figure 2. Argument-dependent name lookup. Here the function name `f` is looked up in the context in which it occurs, and also in the context of its parameter `parm`.

declared in the importing namespace. In the next section, we investigate argument-dependent name lookup, which depends on the relationship between namespaces.

2.3. Argument-dependent name lookup

Figure 2, taken from [12, Section 3.4.2], contains an instance of *argument-dependent name lookup*, where the unqualified name `f` on line 7 of the figure must be looked up in the usual contexts as well as the namespace that contains the argument's type. Visual inspection of the function call at line 7, and the namespace running from lines 1 to 4, shows that name lookup for `f` on line 7 should match the declaration of `f` on line 3.

To provide argument-dependent name lookup for the C++ language, the name lookup facility in the corresponding parser front-end must potentially return a set of declarations. The set of declarations found in a search for a name, such as `f` in the above example, is the union of those found using ordinary unqualified lookup and the set of declarations found in the namespaces and classes associated with the argument types [12, Section 3.4.2].

We describe our solution to the problem of argument-dependent name lookup in Section 4.3.

2.4. Name lookup in *keystone*

Figure 3 summarizes the design of our system to construct a parser front-end for ISO C++. The figure presents two subsystems, illustrated as tabbed folders and designated by the ≪subsystem≫ stereotype. The ProgramProcessor subsystem is shown on the left-hand side and the Symbol Table subsystem is shown on the right-hand side of Figure 3. The ProgramProcessor and Symbol Table subsystems are elaborated in Section 4.

The ProgramProcessor subsystem includes a Scanner and Parser and is responsible for initiating and directing symbol table construction and name lookup. This responsibility includes two phases: (1) assembling the necessary information for creation of a NameOccurrence object; and (2) directing the search for a corresponding NameDeclaration object in the Symbol Table subsystem.

The NameOccurrence object encapsulates local information relevant to the lookup, including the String representation of the name, a Boolean to indicate name qualification (by class or namespace),

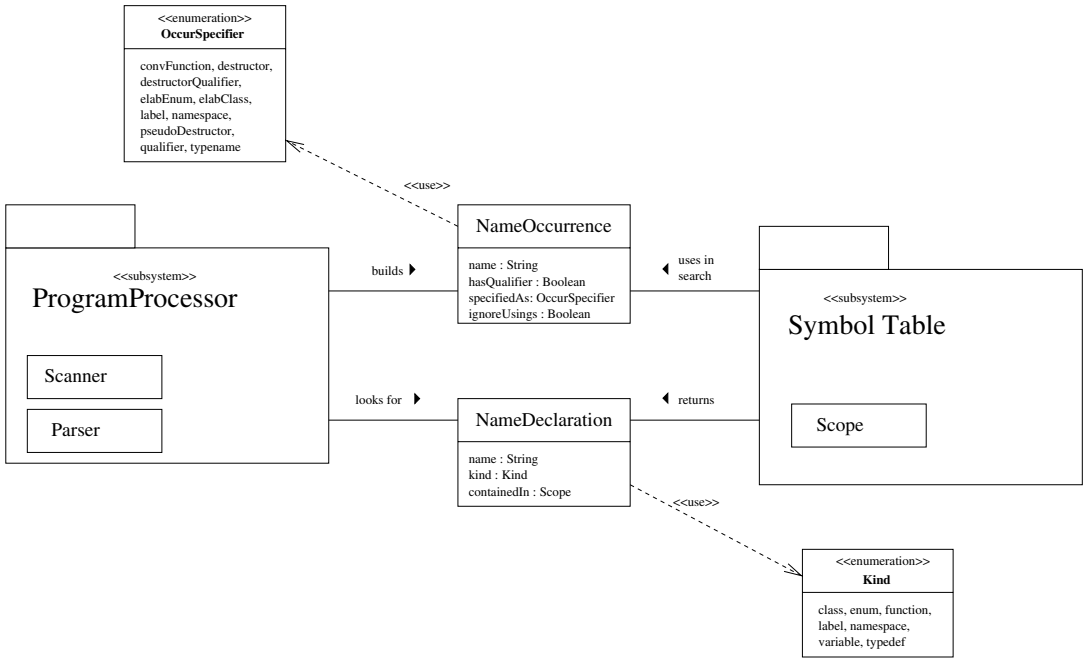

Figure 3. A summary of the system design of our parser front-end for ISO C++. The ProgramProcessor subsystem is responsible for initiating and directing symbol table construction and name lookup by marshaling information about the name in a NameOccurrence object and directing the search for a corresponding NameDeclaration in the Symbol Table subsystem.

and an enumeration, OccurSpecifier, that captures lexical information about the context in which the name occurred. The NameDeclaration object includes the String representation of the name, an enumeration indicating the type of name and a pointer to the enclosing scope.

3. OVERVIEW OF TOKEN DECORATION

In this section we describe our approach to token decoration. In Section 3.1 we present an overview of the approach and in Section 3.2 we describe the technique as implemented in *keystone* [8,11,18], a parser and front-end for ISO C++ [12, Appendix A].

3.1. Token decorated parsing

Figure 4 illustrates the usual interaction between a Scanner, represented by the ellipse on the left, and a Parser, illustrated by the ellipse on the right, during the early phase of compilation. In the figure,

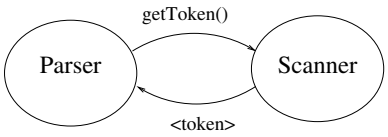

Figure 4. Classic parsing. The usual relationship between the Scanner and Parser. The Parser first requests a token from the Scanner, and the Scanner then responds by sending a token to the Parser.

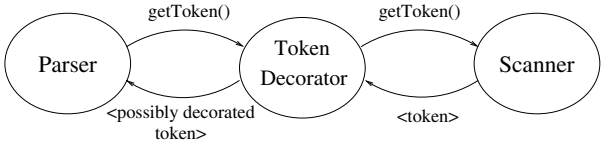

Figure 5. A summary of our approach to token-decorated parsing, with the Parser illustrated on the left, the Scanner on the right and a Token Decorator interposed between the two. The Parser requests a token from the TokenDecorator, which then requests a token from the Scanner. The Scanner returns a token to the TokenDecorator, which then decorates certain tokens and passes either a <token> or a <*possibly decorated token*> to the Parser.

the Parser requests a token from the Scanner by the call to getToken(), and the Scanner responds by passing a token, <token>, to the Parser, which then continues the parse.

Figure 5 illustrates the interaction between a Scanner and Parser using token decoration, where a TokenDecorator, represented by an ellipse, is interposed between the Parser and Scanner to represent the subsystem that decorates tokens. During token decoration, the Parser requests a token from the TokenDecorator rather than the Scanner; this request for a token is represented in the figure by the call to function getToken() on the arrow from the Parser to the TokenDecorator ellipse. The TokenDecorator then passes the message through to the Scanner, which returns the token to the TokenDecorator, where certain tokens are decorated before being passed to the Parser. As Figure 5 illustrates, the Scanner passes a <token> to the TokenDecorator, and the TokenDecorator passes a <possibly decorated token> to the Parser.

In the next section, we demonstrate the technique as implemented in *keystone*, a parser and front-end for ISO C++, including an explanation of our resolution of the *declaration/expression* and *template* ambiguities presented in Section 2.

3.2. Token decoration in *keystone*

To facilitate recognition of ambiguous constructs in C++, we decorate the identifier token in the grammar found in the C++ standard [12, Appendix A]. The original token, IDENTIFIER, may be decorated as:

1. *ID_typedef_name*;
2. *ID_original_namespace_name*;


```
1       type_name
2         : class_name
3         | enum_name
4         | typedef_name
5         ;
6       typedef_name
7         : ID_typedef_name
8         ;
```

Figure 6. The grammar productions for a typedef declaration with the IDENTIFIER token decorated as *ID_typedef_name* on line 7.

```
1       function_definition
2         : decl_specifier_seq_opt declarator
3           ctor_initializer_opt function_body
4         ;
5       declarator
6         : direct_declarator
7         ;
8       direct_declarator
9         : declarator_id
10        | direct_declarator LEFTPAREN
11          parameter_declaration_clause
12          RIGHTPAREN cv_qualifier_seq_opt
13          exception_specification_opt
14        ;
15      declarator_id
16        : id_expression
17        ;
18      id_expression
19        : unqualified_id
20        | qualified_id
21        ;
22      unqualified_id
23        : IDENTIFIER
24        ;
```

Figure 7. The grammar productions for the declaration of a function where the IDENTIFIER token, listed on line 23, is not decorated.

3. *ID_namespace_alias*;
4. *ID_enum_name*;
5. *ID_class_name*; or
6. *ID_template_name*.

We now apply token decoration to disambiguate the *declaration/expression* example discussed in Section 2. Recall that the statement in Equation (1) might be, among other things, (1) a declaration of `x` of type **int** or (2) a function call. To disambiguate the statement, we first perform name lookup on `f`, which is returned from the scanner as an IDENTIFIER token. If lookup determines that `f` is a typedef, then the IDENTIFIER token is decorated to be an *ID_typedef_name* token.

To see how a decorated token can facilitate disambiguation of the parse of Equation (1), consider the productions in Figure 6, taken from the C++ standard [12, Appendix A], with IDENTIFIER replaced by *ID_typedef_name*. Assume that `f` is a typedef. When the bottom-up Parser requests a token from the TokenDecorator, a decorated token, *ID_typedef_name*, is returned. The *ID_typedef_name* is reduced, on line 7 of Figure 6, to a typedef_name on line 6; the typedef_name on line 4 is then reduced to a type_name, and the statement in Equation (1) is correctly parsed as a declaration.

To illustrate recognition of Equation (1) as an expression, assume that `f` is a function call. In this case the TokenDecorator returns the normal IDENTIFIER token, which is reduced, on line 23 of Figure 7, to an unqualified_id. The unqualified_id is reduced to an id_expression on line 18, to a declarator_id on line 15, to a direct_declarator on line 8, to a declarator on line 5 and finally is used in the production for a function on line 2 of Figure 7.

4. NAME LOOKUP IN *keystone*

In this section, we provide details about our implementation of name lookup, the *sine qua non* of token-decorated parsing. The TokenDecorator uses symbol table information to determine if the IDENTIFIER token should be decorated and which of the decorated tokens to return to the parser. The technique of decorating tokens can be applied in any approach to parsing where the scanner and parser work in tandem, as described in Section 3, and can be applied to top-down or bottom-up parsing [19,20]. Our implementation of name lookup is in *keystone*, and our focus on name lookup is on its use in token decoration.

4.1. Implementation details

After receiving a request for a token from the Parser, the TokenDecorator sends a request to the Scanner. If the token is an IDENTIFIER, the TokenDecorator requests name information from the symbol table to determine if the token should be decorated. Figure 8 illustrates the important *keystone* classes involved in token decoration. In the figure, the parser, Parser, is represented by the rectangle on the left and the scanner is represented by the rectangle on the right marked Scanner; *keystone* uses *btyacc* and *flex*, for back-track parsing and scanning respectively, and *keystone* is implemented in C++. To perform token decoration, *keystone* must buffer some tokens; the buffer is represented by the rectangle labeled TokenBuffer in Figure 8 and is discussed further in Section 4.2.

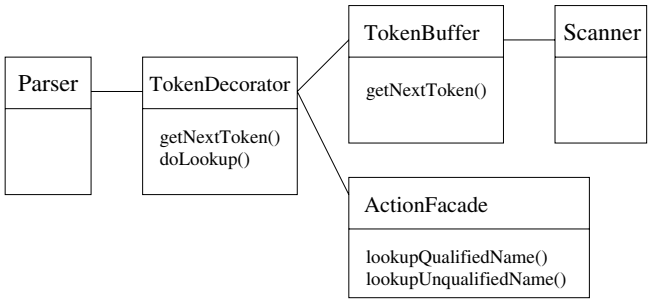

Figure 8. The important classes in the ProgramProcessor subsystem, discussed in Section 2.

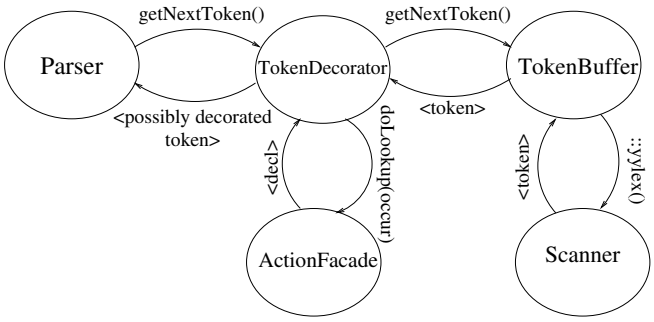

Figure 9. A state chart that summarizes the actions of the *keystone* front-end in token-decorated parsing.

To perform name lookup on an IDENTIFIER, the TokenDecorator consults the ActionFacade, represented by the rectangle on the lower right of Figure 8. The classes illustrated in Figure 8 represent the ProgramProcessor subsystem, discussed in Section 2 and illustrated in Figure 3; the ActionFacade uses the Facade design pattern to abstract symbol table details away from the ProgramProcessor [15]. Details about symbol table representation and name lookup, as abstracted by the ActionFacade, can be found in [8] and [18].

Figure 9 is a state diagram that simulates the actions of *keystone* in token-decorated parsing, with the Parser depicted in the ellipse on the left and the Scanner depicted in the ellipse on the lower right of the figure. The Parser sends the message getNextToken() to the TokenDecorator, which passes the message through to the TokenBuffer. The TokenBuffer requests a token from the Scanner, which passes <token> back to the TokenBuffer, which passes <token> back to the TokenDecorator. Finally, the TokenDecorator constructs a NameOccurrence, occur, and asks the ActionFacade to perform name lookup, doLookup(occur), on the name occurrence. The ActionFacade returns either a


```
( 1)  int TokenDecorator::getNextToken() {
( 2)    int currentToken = buf.getNextToken();
( 3)    if ( currentToken == IDENTIFIER ) {
( 4)         NameOccurrence * occur = buf.getNameOccurrence();
( 5)         NameDeclaration * decl = doLookup(occur);
( 6)         if ( decl == NULL ) {
( 7)              decl = new NameDeclaration(occur, Kind::Variable,
( 8)                        facade.getCurrentScope());
( 9)          }
(10)          delete occur;
(11)          yylval.decl = decl;
(12)          currentToken = getContextSensitiveIdent(decl);
(13)     }
(14)     return currentToken;
(15)  }
(16)  NameDeclaration* TokenDecorator::doLookup(
(17)         NameOccurrence* id) const {
(18)    NameDeclaration * decl = NULL;
(19)    if ( id→hasQualifier() ) decl = facade.lookupQualifiedName(id);
(20)    else decl = facade.lookupUnqualifiedName(id);
(21)    return decl;
(22)  }
(23)  int TokenDecorator::getContextSensitiveIdent(
(24)         NameDeclaration * decl) const {
(25)    switch (decl→getKind()) {
(26)         case (Kind::Typedef) : return ID_Typedef_name;
(27)         case (Kind::Class) : return ID_Class_name;
(28)         . . .
(29)         default : return IDENTIFIER;
(30)     }
(31)  }
```

Figure 10. Algorithm getNextToken An outline of the C++ code for getNextToken(), which choreographs the steps in decorating tokens. We also include auxiliary functions doLookup() and getContextSensitiveIdent().

NameDeclaration, shown as <decl> in Figure 9, or NULL to indicate that the name occurrence was not found in the symbol table.

The actions of the TokenDecorator are further detailed in Figure 10, which contains three C++ functions in the TokenDecorator class of *keystone*: getNextToken(), doLookup() and getContextSensitiveIdent().

The steps in token decoration are choreographed by getNextToken(), listed on lines 1 through 15 of Figure 10. On line 2, a token is requested from the token buffer and placed in the temporary

currentToken. On line 3 currentToken is tested and if it is an identifier it may be decorated during lines 4 through 13. On line 4 of Figure 10 a name occurrence is requested and used in name lookup, initiated on line 5 of the figure.

Name lookup is continued on line 16 in function doLookup, where qualified name lookup is initiated on line 19 and unqualified name lookup is initiated on line 20. In this section we are investigating name lookup for unqualified identifiers such as f in Equation (1); we consider qualified name lookup in the next section. doLookup returns NULL if an identifier is not found in the symbol table and a new NameDeclaration is constructed on line 7 with the type of this identifier marked as a variable, Kind::Variable. On line 12 the function getContextSensitiveIdent is called to possibly decorate the token.

If the identifier is newly created on line 7 of getNextToken(), then on line 29 of getContextSensitiveIdent the default case of the **switch** will be chosen and an undecorated token, IDENTIFIER, will be returned to getNextToken(). However, if the **switch** matches Kind::Class, Kind::Enum, Kind::Namespace, Kind::Typedef or Kind::Template, then a decorated token is returned to getNextToken(). Either the decorated or non-decorated token is returned to the parser on line 14 of Figure 10.

4.2. Token buffering in qualified name lookup: the left context

Each name in the program is represented in the *keystone* symbol table by an instance of NameDeclaration, which stores information about the name such as its current scope, its enclosing scope and its qualifier, if it is a qualified name[§]. To facilitate lookup of names in the program, *keystone* constructs an instance of NameOccurrence that marshals information about the name; this information is garnered from the current context of the name as captured in the token buffer, TokenBuffer in Figure 8.

A snapshot of a token buffer is illustrated in Figure 11, where namespaces A, B and C are shown with variable x local to namespace C. A qualified reference to x, A::B::C::x = 0; is shown below the namespaces in the figure and below that, a snapshot of the token buffer containing five tokens. We have found that five tokens is all that is required to capture the context of the current token under consideration: the current token is in the middle of the buffer, the two tokens to the left of the current token capture the left context and the two tokens to the right of the current token capture the right context.

We now discuss the lookup of qualified name x, local to namespace C in Figure 11. When a name occurrence is requested on line 4 of Figure 10, fields in the occurrence are marked to indicate the qualifier, in this case namespace C; the token buffer marks the name occurrence and it uses the context to the left of currentToken to determine qualification. The context of currentToken is illustrated at the bottom of Figure 11. Name lookup is initiated on line 5 of Figure 11 and continued in doLookup, using the occur parameter to guide lookup. On line 19, the name occurrence is tested for qualification and, if qualified, the lookup is passed on to the action facade, where a qualified name lookup will be initiated in the symbol table. The search for x in the symbol table will be conducted in the context of namespace C; a pointer to C is included as a data attribute of the name occurrence used in the lookup process.

[§]Please see Section 2.2 for information about qualified names in C++.

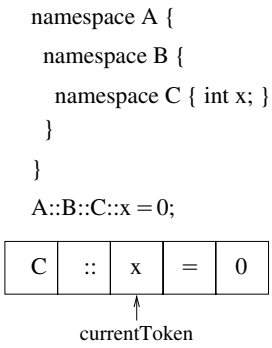

Figure 11. Our use of buffered tokens to facilitate name lookup in *keystone*. The variable currentToken points to the current token; the two tokens to the left of currentToken represent the left context and the two tokens to the right of currentToken represent the right context.

4.3. Argument-dependent name lookup

Argument-dependent name lookup was described in Section 2.3 and an example presented in Figure 2. In the figure, name lookup of f on line 7 must include a search of the scopes for ordinary name lookup of f and the set of declarations associated with the classes or namespaces of any function arguments of f that are qualified by a class or namespace. We now describe our implementation of argument-dependent name lookup in *keystone*.

To implement argument-dependent name lookup on a function name such as `f`, we perform name lookup twice for a function that has arguments that are qualified by a class or namespace. The first lookup is initiated by the token decorator in the usual way, and the second lookup is initiated by the parser to incorporate argument-dependent name lookup.

The first time that the parser encounters `f`, lookup fails, the identifier is not decorated and `f` is parsed correctly as a function call. Thus, token decoration works correctly even though name lookup initially fails. However, to correctly install the name of an argument-dependent function in the symbol table, name lookup must be performed a second time. The second name lookup occurs when the parser reaches the end of the function argument list, and the parser initiates a second lookup through the ActionFacade, passing the list of arguments for the function to the facade. The facade then initiates a second lookup of the function name, using the argument list to determine if the declaration might be found in an associated namespace. If the parameter is a qualified type, the associated namespace is searched and included in the set of possible declarations for the function name.

5. DIFFICULTIES ENCOUNTERED IN TOKEN-DECORATED PARSING

One of the difficulties in token-decorated parsing is that the token decorator must be able to determine if a token should be decorated. In the case that the token decorator cannot yet perform name lookup,


```
( 1)  class A {
( 2)    A () { }
( 3)    int f(const A & a) {}
( 4)  };
```

Figure 12. Recognizing class type argument. A member function with a class type argument. Name lookup on the argument may match the argument with the class constructor, mistakenly recognizing the argument as a function type rather than a class type.

such as in argument-dependent name lookup, the token decorator must be designed so that it returns the correct token or that the default token that is returned is correct. Thus, in our *keystone* front-end, the token decorator must be able to perform name lookup for each identifier in the program. If name lookup is unsuccessful or returns the wrong declaration to match an identifier, then the program may not parse correctly.

In this section, we describe one of the difficulties encountered in our implementation of name lookup in *keystone*. The first difficulty relates to the type of a function argument and the second relates to scope issues for data initialization in constructors.

5.1. Recognizing class parameters

Consider Figure 12, containing a class declaration with a constructor, on line 2, and a member function `f` on line 3. In parsing line 3, the identifier `f` is not found in the symbol table, `f` is not decorated and an IDENTIFIER token is correctly returned from the TokenDecorator to the parser, as explained in Section 3.2.

A potential problem occurs when the TokenDecorator receives the IDENTIFIER token `A` from the token buffer and then performs name lookup on `A`. The most recent declaration of `A` in the symbol table may match the constructor on line 2 in Figure 12; in this case the IDENTIFIER is not decorated and the parser does not recognize the argument type, `A`, for function `f` on line 3 as a class type argument. Thus, the parser may incorrectly parse the program.

To correct this problem, class constructors are stored in the *keystone* symbol table in the form <class-name>::<class-name>. This solution is consistent with the C++ standard [12, Section 12.1.1]. For the example in Figure 12, the constructor is stored in the symbol table with the name $A :: A$. Thus, when the class type argument is encountered on line 3 by the token decorator, name lookup correctly returns `A` as a class type, since the most recent declaration of `A` is the class declaration on line 1. Using this solution, the parser will choose the correct derivation for the function prototype.

5.2. Constructor member initialization

In this section we discuss a second situation where name lookup may cause a problem for token decoration. Figure 13 illustrates a class `A` using member initialization for data attributes `i` and `j`. The prototype for the constructor is completely parsed when the parser encounters the initialization of `i` and `j`. If the tokens `i`, `j` or `k` needed to be decorated, it would be difficult for name lookup to find the


```
( 1)  class A {
( 2)    A ( int k ) : i(k), j(k) { }
( 3)    int i, j;
( 4)  };
```

Figure 13. Scope of the initializer. In this example, the use of k as an initializer should be matched with the parameter k, even though it appears to be occurring in the scope of the class A itself.

variable k because, since the prototype has been completely parsed, name lookup will be performed in the scope of the class A. Thus, name lookup will not find the variable k and will have difficulty decorating it.

6. RESULTS

In this section we describe the results of our study of token decoration in *keystone*, our parser front-end for the ISO C++ language. In Section 6.1 we describe the test suite for the study and in Section 6.2 we measure the number of tokens decorated in the test suite. In Section 6.3 we use *keystone* and six other compilers to parse test cases from Clause Three of the ISO standard.

The experiments in this section were executed on systems running version 7.1 of Red Hat Linux and Solaris SunOS version 5.8. To provide some insight into the efficiency of *keystone*, we were able to parse the ep matrix application, the test case with the most tokens, in 11.74 s on a Dell Precision 530 workstation, with a Xeon 1.7 GHz processor and 512 MB of RDRAM, running the Red Hat Linux 7.1 operating system. Our implementation language was C++ compiled with GNU *gcc* version 2.96. The *keystone* front-end is implemented with 3135 lines of C++ code and the *btyacc* grammar has 527 grammar rules, 124 terminals and 218 non-terminals.

6.1. The test suite

Table I summarizes our suite of seven test cases, listed in the rows of the table as encrypt, Clause 3, php2cpp, fft, graphdraw, ep matrix and vkey. The test cases in the suite were chosen because of their range and variety of application and they are listed in sorted order by number of lines of code, not including comments or blank lines.

Test case encrypt is an encryption program that uses the Vignere algorithm [21] and Clause 3 is a sequence of examples taken from Clause 3 of the ISO C++ standard [12]. The Clause 3 test case includes intricate examples of name lookup, including argument-dependent name lookup, described in previous sections; we discuss and use the Clause 3 test case in the next section to compare *keystone* with six other compilers [22]. The php2cpp test case [23] converts the PHP Web publishing language to C++ and fft performs fast Fourier transform [24]. graphdraw [25] is a drawing application that uses *IV Tools* [26], a suite of free X Windows drawing editors for PostScript, TeX and Web graphics production. The ep matrix test case is an extended precision matrix application that uses *NTL*, a high-performance portable C++ number theory library providing data structures and algorithms for manipulating signed

Table I. A summary of the information about the number of lines, the number of classes, the number of classes with functions and the number of namespaces in each of the seven test cases.

Test case	Lines	Classes	Classes w/ fns	Namespaces
encrypt	946	1	1	0
Clause 3	952	40	34	63
php2cpp	1920	6	6	0
fft	2238	51	36	0
graphdraw	4354	199	76	0
ep matrix	4944	78	51	0
vkey	8556	279	44	0

arbitrary length integers, and for vectors, matrices and polynomials over integers and finite fields [27]. vkey [28, p. 760] is a GUI application that uses the *V GUI* library [29], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications.

The columns of Table I list details about the number of lines of code, not including comments or blank lines, the number of classes, the number of classes with functions and the number of namespaces for each of the test cases. All of the test cases are complete programs, except Clause 3, and three use large libraries: ep matrix, vkey and graphdraw use the *NTL*, *V GUI* and *IV Tools* libraries, respectively.

The third and fourth columns in Table I compare the number of classes with the number of classes that have functions. We make this distinction to distinguish between classes used as old 'C-style' structs, and proper classes. Since the ep matrix, vkey and graphdraw test cases use libraries, including a larger number of system library files than the other test cases, many of the classes are actually structs that contain only data and no functions. The data in the sixth row, fourth column in the table show that the vkey test case, for example, has 279 classes. However, only 44 of the 279 classes in vkey have functions and most of the remaining 235 classes are structs containing data and no functions.

Finally, only the Clause 3 test case contains namespaces, as shown in the second row, last column of the table. The recent inclusion of namespaces into the standard, with their concommitant recent use in textbooks and by programmers, is reflected here in the test suite. The Clause 3 test case is a sequence of examples taken directly from the ISO standard where the semantics of name lookup for namespaces is specified.

6.2. Number of tokens decorated

Figure 14 provides some results about the number of tokens that are decorated by *keystone*. The rows of the table list the test cases and the columns list the data acquired by monitoring the token buffer and token decorator subsystems, including the number of tokens, Tokens, the number of identifiers in the test cases, Id's, the number of decorated identifiers, Id's decorated, the average number of decorated

Test case	Tokens	Id's	Id's decorated	Avg tokens decorated (%)	Specifiers	Qualifiers
encrypt	7593	1638	445	5.8	151	33
Clause 3	2563	629	171	6.6	270	59
php2cpp	10 942	2517	306	2.7	62	0
fft	14 297	3604	818	5.7	226	37
graphdraw	23 821	6188	2463	10.3	641	123
ep matrix	51 927	17 300	5260	10.1	706	65
vkey	29 914	8541	3477	11.6	953	0

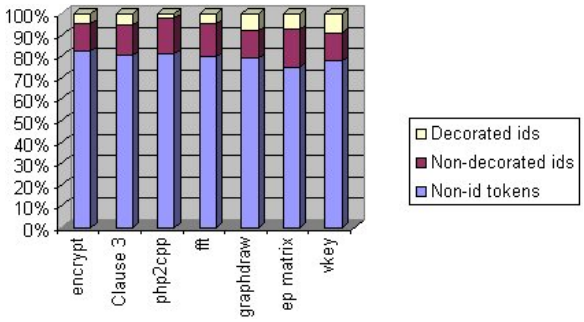

Figure 14. The table shows the number of tokens, identifiers and decorated tokens in each of the test suites. These figures are presented in summary form in the graph. Also shown in the table are the numbers of specifiers and qualifiers, which directly impact the name lookup problem.

tokens, Avg tokens decorated, the total number of specifiers used in the name occurrences during name lookup, Specifiers, and the number of qualified identifiers, Qualifiers, found in each of the test cases.

The fifth column in the table of Figure 14 details the average number of tokens decorated and the graph in the figure summarizes these results, with a bar representing information for each test case. Each bar is tri-colored: the top portion of each bar is the percentage of decorated identifiers, the middle portion of each bar is the percentage of non-decorated identifiers and the bottom portion of each bar is the percentage of non-identifier tokens for each of the test cases. We listed the kinds of tokens that are decorated in Section 3.2, and the top portion of each bar represents the values in column 5 of the table and reflects the percentage of tokens in the test case that are decorated by *keystone*. The graph illustrates that php2cpp required the lowest percentage of tokens to be decorated, while vkey required the highest percentage of tokens to be decorated, reflecting their respective usage of tokens such as typedefs, enums and constructors. Our study is ongoing; however, token decoration as implemented in *keystone* enables disambiguation of grammar derivations without requiring many tokens to be decorated.

6.3. Conformance study

One of the goals of our work is to use the grammar from the ISO standard to build a parser front-end for C++ that conforms to the standard. We exploit token decoration to obviate modification or

Table II. Conformance study. Here we compare *keystone* against six other compilers for conformance with the ISO standard. For this test, we used 56 positive test cases from Clause Three of the ISO C++ standard, and we report above on the number of failures.

	keystone	*gcc* 3.0.4	*gcc* 2.96	*gcc* 2.95.2	MIPSpro 7.3.1.2m	Borland 5.5.1	VC++ 6.0
failed	3/56	6/56	6/56	6/56	7/56	6/56	13/56

transformation of the grammar. In Table II we summarize the results of our comparison of *keystone* with six other compilers, including the GNU collection of *gcc* 3.0.4, *gcc* 2.96 and *gcc* 2.95.2, MIPSpro 7.3.1.2m, Borland 5.5.1 and Microsoft VC++ 6.0.

In order to apply the same standard to all of the compilers under consideration, we use the same test cases and the same testing framework for all executions, even though some of the compilers are platform dependent and there is no common platform for all compilers. We found that the Python language provided the functionality that we needed with its scripting facility, its platform independence and its object orientation to facilitate code reuse [30]. Moreover, unlike any other language, Python includes a testing framework as a module of the language. This testing framework is *PyUnit*, patterned after the JUnit framework developed by Gamma and Beck [31], and included as a Python module, *unittest* [32], in Python versions 2.1 and later [30]. The interested reader may find a more thorough presentation of C++ compiler conformance in [22].

We have extended the *PyUnit* framework to facilitate measurement of ISO conformance and we use the extended framework in all test case executions. To avoid bias for or against any compiler, our test case selection is based on examples found directly in Clause Three of the ISO C++ standard [12], which contains 88 code samples that we translated into 56 positive test cases and 32 negative test cases. The outcomes for the examples are specified in the standard and the results of using the 56 positive test cases to test *keystone* and the other six compilers are illustrated in Table II.

For the results in Table II, we compiled the 56 test cases but did not link or execute them, since we are only interested in the parsing phase of compilation. The rows of the table indicate that *keystone* failed to parse three of the test cases, the GNU collection each failed to parse six, MIPSpro 7.3.1.2m failed to parse seven, Borland 5.5.1 failed to parse six and Microsoft VC++ 6.0 failed to parse 13 of the test cases. The results reported in Table II do not include certain semantic checks nor negative test cases; nevertheless, the results indicate that our exploitation of token decoration to obviate modification of the grammar can facilitate conformance to a standard.

7. RELATED WORK

The C++ programming language was developed directly from C and, as mentioned earlier, inherits some of its context sensitivity from this language. However, it is notable that the standard reference for C [33] presents a grammar that requires only a distinction between *identifier* and *typedef-name*

in order to be acceptable to the *yacc* parser generator (with only the trivial dangling-else ambiguity). The extra complexity of the scoping structure of early versions of C++ were noted in [34], which proposed 'flexible symbol table structures' as a solution.

One of the earliest and most-cited grammars for C++ is that of Jim Roskind [10]. However, this grammar is based on an early version of the language as described in [35] and does not include exceptions, namespaces or templates. This grammar, containing 42 conflicts when processed by *Berkeley yacc* (v 1.8) also requires a distinction between *identifier* and *typedef-name*. Roskind describes this as a 'feedback loop' between the parser and scanner, and refers to the necessitated changes to the scanner as a 'lex hack', noting that the flow of information between the parser and the scanner must be swift in order to ensure correctness. Interestingly, he blames the syntactic difficulties for the non-compliance of many then-current compilers with the specification from [35].

The object-oriented front-end for C++ described in [9] uses an elaborate system of token decoration, distinguishing ordinary identifiers, type names, enum names, template names and constructors. Context-sensitive token mutation takes place for four other non-identifier tokens, along with the insertion of nine special purpose tokens to aid disambiguation. In addition, while the overall parser is *yacc*-based, a small-recursive descent parser is used to properly identify declarations. However, this grammar has no support for namespaces and the more recent template features such as explicit specialization and instantiation.

The *sage++* system [5] provides a unified framework for parsing C, Fortran and C++, and extensive documentation on the abstract syntax trees that can be generated, but as with the other parsers it has difficulty with namespaces and some template features. More recent work includes an LL(1) parser [7]; however, despite modifying the PCCTS parser-generator specifically for this project, the work seems incomplete. One interesting feature is the use of arbitrary LL(k) lookahead in order to disambiguate declarations.

An interesting analysis of the ambiguities for modern C++ is given in [6], where a system is presented that attempts to parse C++ programs even when corresponding header files are missing. Despite relatively high success rates, their parser is necessarily based on heuristics, and thus cannot, in general, deliver an accurate parse. Another approach, described in [36] is to deterministically parse a superset of C++, and then apply filters in subsequent passes to rule out spurious cases.

The other main approach to parsing C++ is the use of GLR parsers, originally described in [37], and developed in [38–40]. In this approach, ambiguities are not deterministic but instead all possible paths are recorded; thus the end product of a parse is a graph, rather than a tree. Subsequent filters can then be applied to eliminate remaining ambiguities. This approach is advocated for languages such as C++ in [41], and for software renovation in [42]. However, there does not yet appear to be a publicly-available implementation of an ISO C++ parser using this technique.

The Edison Design Group (EDG) [43] provides a C++ front-end that processes ISO C++ and generates a representation of the program in the C language. The EDG front-end is highly reliable and has been used in a number of complex applications. EDG is written using 315 000 lines of commented C-code and is a heavyweight, sophisticated commercial product. Our goal in the design and implementation of *keystone* is to provide a freely-available minimal C++ parser that can be used as a kernel for more complex applications. We believe that the simplifications arising from token decoration, along with the object-oriented design of our system, facilitate this goal. Our future work includes a comparison of EDG with *keystone* to measure conformance to the ISO standard.

8. CONCLUDING REMARKS

We have described the use of token decoration to facilitate the parse of ambiguous constructs such as the *declaration/expression* and *template/less-than* ambiguity. Also, we have shown that token declaration coupled with token buffering can facilitate name lookup for qualified names and argument-dependent lookup in C++. We have implemented token decoration and token buffering in *keystone*, a parser front-end for ISO C++, and we have shown that, in practice, few tokens are decorated. Finally, we have compared *keystone* to other parsers and shown that our approach of using the grammar from the C++ standard, together with token decoration to obviate modification of the grammar, can produce a parser front-end that passes more test cases extracted from Clause Three of the standard than the other parsers that we tested.

We are currently exploiting *keystone* in several on-going projects. We have constructed an application programmer interface, *Clouseau*, that facilitates easy inspection and extraction of information in the *keystone* symbol table [16]. We have used *keystone* and *Clouseau* to reverse engineer UML class diagrams [16] and to construct a *taxonomy of classes* to identify the best implementation-based approach to testing [17]. We have also used the taxonomy of classes to measure the changes in software across different releases [44].

Keystone is freely available under the GNU license. The source may be obtained either from *SourceForge* at http://sourceforge.net/projects/keystone-ccs or from the author's Web page at http://www.cs.clemson.edu/~malloy/projects/keystone/keystone.html.

REFERENCES

1. Cohen SF. Quest for Java. *Communications of the ACM* 1997; **41**(1):81–83.
2. Singhal S, Nguyen B. The Java factor. *Communications of the ACM* 1998; **41**(6):34–37.
3. Tyma P. Why are we using Java. *Communications of the ACM* 1998; **41**(6):38–42.
4. Sutter H. C++ conformance roundup. *C/C++ User's Journal* 2001; **19**(4):3–17.
5. Bodin F, Beckman P, Gannon D, Gotwals J, Narayana S, Srinivas S, Winnicka B. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. *The Second Annual Object-oriented Numerics Conference (OON-SKI)*, Sunriver, OR, 1994; 122–136.
6. Knapen G, Lague B, Dagenais M, Merlo E. Parsing C++ despite missing declarations. *7th International Workshop on Program Comprehension*, Pittsburgh, PA, 5–7 May 1999. IEEE Computer Society: Los Alamitos, CA, 1999.
7. Lilley J. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.
8. Power JF, Malloy BA. Symbol table construction and name lookup in ISO C++. *37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2000)*, Sydney, Australia, November 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 57–68.
9. Reiss SP, Davis T. Experiences writing object-oriented compiler front ends. *Technical Report*, Brown University, January 1995.
10. Roskind JA. *A YACC-able C++ 2.1 Grammar, and the Resulting Ambiguities*. Independent Consultant: Indialantic, FL, 1989.
11. Power JF, Malloy BA. Metric-based analysis of context-free grammars. *Proceedings 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000. IEEE Computer Society: Los Alamitos, CA, 2000.
12. American National Standards Institute. *International Standard: Programming Languages—C++*. Number 14882:1998(E) in ASC X3. ISO/IEC JTC 1, September 1998.
13. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide* (*Object Technology Series*). Addison-Wesley: Reading, MA, 1999.
14. Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, Inc., 1999.
15. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.

16. Matzko S, Clarke P, Gibbs TH, Malloy BA, Power JF, Monahan R. Construction of UML class diagrams for C++. *40th International Conference on The Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society: Sydney, Australia, 2002.
17. Clarke P, Malloy BA. A unified approach to implementation-based testing of classes. *Proceedings of 1st Annual International Conference on Computer and Information Science (ICIS '01)*, Orlando, FL, 3–5 October 2001. ACIS: Mt. Pleasant, MI, 2001.
18. Power JF, Malloy BA. An approach for modeling the name lookup problem in the C++ programming language. *ACM Symposium on Applied Computing*, Como, Italy, March 2000. ACM: New York, NY, 2000.
19. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques and Tools*. Addison-Wesley: Reading, MA, 1986.
20. Muchnick SS. *Advanced Compiler Design & Implementation*. Morgan-Kaufman: San Francisco, 1997.
21. Alexander S. The C++ resources network. http://www.cplusplus.com [October 2001].
22. Malloy BA, Linde SA, Duffy EB, Power JF. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal* June 2002; 71–80.
23. Cavalier FJ. Debugging PHP using a C++ compiler. *Dr. Dobbs Journal* March 2002; 42–46.
24. Kiselyov O. Fast Fourier transform. Free C/C++ Sources for Numerical Computation. http://cliodhna.cop.uop.edu/~hetrick/c-sources.html [March 2002].
25. Vlissides JM, Linton MA. Applying object-oriented design to structured graphics. *Proceedings of USENIX C++ Conference*, Denver, CO, October 1988. USENIX, 1988.
26. Vlissides JM, Linton MA. IV tools. http://www.vectaport.com/ivtools/ [March 2002].
27. Shoup V. Number theory library. http://www.shoup.net/ntl/ [March 2002].
28. Swan T. *GNU C++ for Linux* (1st edn). Que Corporation, a Division of Macmillan: Indianapolis, IN, 2000.
29. Wampler B. The V C++ GUI framework. http://www.objectcentral.com [October 2001].
30. van Rossum G. *Python Library Reference*. Python Software Foundation, 2001.
31. Gamma E, Beck K. Test infected: Programmers love writing tests. http://members.pingnet.ch/gamma/junit.htm [2001].
32. Purcell S. Python unit testing framework. http://pyunit.sourceforge.net/ [March 2002].
33. Kernighan BW, Ritchie DM. *The C Programming Language* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 1988.
34. Dewhurst SC. Flexible symbol table structures for compiling C++. *Software—Practice and Experience* 1987; **17**(8):503–512.
35. Ellis MA, Stroustrup B. *The Annotated C++ Reference Manual*. Addison-Wesley: Reading, MA, 1990.
36. Willink ED. Meta-compilation for C++. *PhD Thesis*, Computer Science Research Group, University of Surrey, June 2001.
37. Tomita M. *Efficient Parsing for Natural Langauges*. Kluwer Academic Publishers: Norwell, MA, 1985.
38. Rekkers J. Parser generation for interactive environments. *PhD Thesis*, University of Amsterdam, 1992.
39. Visser E. Syntax definition for language prototyping. *PhD Thesis*, University of Amsterdam, 1997.
40. Aycock J, Horspool N, Janousek J, Melichar B. Even faster GLR parsing. *Acta Informatica* 2001; **37**(9):633–651.
41. Wagner TA, Graham SL. Incremental analysis of real programming languages. *Conference on Programming Language Design and Implementation*, Las Vegas, NV, 15–18 June 1997. ACM: New York, NY, 1997; 31–43.
42. van den Brand M, Sellink A, Verhoef C. Current parsing techniques in software renovation considered harmful. *International Workshop on Program Comprehension*, Ischia, Italy, 24–26 June 1998. IEEE Computer Society: Los Alamitos, CA, 1998.
43. Edison Design Group. C++ Front End. http://www.edg.com/cpp.html [28 December 2000].
44. Clarke P, Malloy BA. Taxonomy of classes to identify changes during maintenance. *Proceedings of the International Conference on Computer and Information Systems, ICIS'2002*, Seoul, Korea, 8–9 August 2002; 631–636.

